МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Автоматизированные системы управления»

ОСНОВЫ ПРОГРАММНОЙ ИНЖЕНЕРИИ

Методические рекомендации к практическим занятиям для студентов направления подготовки 09.03.04 «Программная инженерия» дневной формы обучения

Могилев 2019

УДК 004.413
ББК 3 2.973-018.2
О 75

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой «Автоматизированные системы управления» «04» сентября 2018 г., протокол № 2

Составитель канд. техн. наук, доц. К. В. Овсянников

Рецензент Ю. С. Романович

Приведены методические рекомендации к практическим работам для студентов направления подготовки 09.03.04 «Программная инженерия» дневной формы обучения по дисциплине «Основы программной инженерии».

Учебно-методическое издание

ОСНОВЫ ПРОГРАММНОЙ ИНЖЕНЕРИИ

Ответственный за выпуск	А. И. Якимов
Технический редактор	С. Н. Красовская
Компьютерная верстка	Н. П. Полевничая

Подписано в печать . Формат 60×84/16. Бумага офсетная. Гарнитура Таймс.
Печать трафаретная. Усл. печ. л. . Уч.-изд. л. . Тираж 31 экз. Заказ №

Издатель и полиграфическое исполнение:
Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№1/156 от 24.01.2014.
Пр. Мира, 43, 212000, Могилев.

Содержание

Введение

Нынешнее состояние науки и техники требует от инженерно-технических и научных работников знания средств вычислительной техники и умения обращения с современными программно-техническими комплексами. Эффективное применение компьютеров для решения инженерных и научных задач невозможно без знаний основных методов составления схем алгоритмов, написания действенного программного обеспечения на языке программирования, использования пакетов программ инженерной графики и математических систем.

Цели практического цикла работ:

– изучение основ специальности;

– рассмотрение технологических основ процесса разработки программного обеспечения;

– проектирование и разработка отдельных элементов системы.

Полученные при изучении дисциплины знания и навыки могут быть востребованы при курсовом проектировании и в дальнейшем процессе обучения студента в вузе.

Практическая работа № 1. Сложность программного обеспечения

Цель работы: изучить проблематику создания сложной программной системы в отношении к разрабатываемой ИС.

План практической работы

Вводная часть.
Задание на практическое занятие: сложность программного обеспечения.
Основная часть.
Список обсуждаемых вопросов:
– трудность управления процессом разработки;
– необходимость обеспечить достаточную гибкость программы;
– способы и описания поведения больших дискретных систем.
– сложность реальной предметной области, из которой исходит заказ на разработку.
Заключительная часть.
Подведение общих итогов занятия.
Ответы на вопросы студентов.
Выдача рекомендаций.

Контрольные вопросы

1 Что такое управление процессом разработки?
2 Что такое гибкость программного обеспечения?
3 Как описывается поведение программных систем?
4 Что такое сложность ПО?

Краткие теоретические сведения

Особенности разработки сложных (больших) программных систем. Из года в год увеличиваются разнообразие и сложность систем, получивших в международной научно-технической практике название систем, интенсивно использующих программное обеспечение, – Software Intensive Systems (SIS). В системах такого рода функциональный потенциал определяется программным обеспечением (ПО) или зависит от ПО в существенной мере.

В таких системах программные компоненты взаимодействуют друг с другом и компонентами и подсистемами другой природы, датчиками, приборами и людьми, вовлеченными в процессы использования SIS. К числу SIS, например, относятся разнородные автоматизированные системы управления, встроенные бортовые транспортные системы, телекоммуникационные и корпоративные системы, в том числе и на базе web-сервисов. Для разработок SIS типичны крупномасштабные проекты – десятки или сотни разработчиков, месяцы или годы

разработки, сотни тысяч или десятки миллионов долларов, комплектование из многочисленных разнородных подсистем, большая часть из которых включает программные системы.

Не все программные системы сложны. Существует множество программ, которые задумываются, разрабатываются, сопровождаются и используются одним и тем же человеком. Обычно это начинающий программист или профессионал, работающий изолированно. Нельзя сказать, что все такие системы плохо сделаны или тем более усомниться в квалификации их создателей. Но такие системы, как правило, имеют очень ограниченную область применения и короткое время жизни. Обычно их лучше заменить новыми, чем пытаться повторно использовать, переделывать или расширять. Разработка подобных программ скорее утомительна, чем сложна, так что изучение этого процесса нас не интересует.

Какого-либо одного формального признака, отличающего обычную программу от сложной, не существует. В целом сложные программы выгодно отличаются разнообразием предоставляемого сервиса и количеством обрабатываемой информации. Возможно обозначить лишь некоторые качественные характеристики, свойственные сложной программе. Сложная программа характеризуется также более сложным алгоритмом обработки событий. В частности, такая программа предполагает некоторую реакцию на вмешательство пользователя в управляемый процесс или объект.

Существенно, что сложные программы предназначены для многократного использования и применения разными пользователями. В связи с этим следует обратить внимание на ряд необходимых свойств программного обеспечения.

Обычно сложная программа обладает следующими свойствами:

– программа решает одну или несколько связанных прикладных задач, зачастую сначала не имеющих четкой постановки и настолько важных для каких-либо лиц или организаций, что те приобретают значимые выгоды от ее использования;

– программа не предназначена для решения каких-либо прикладных задач, но от нее зависит эффективное решение этих прикладных задач. Это системные программы, например операционные системы, системы управления базами данных, различные инструментальные системы и т. п.;

– существенно, чтобы программа была удобной в использовании. В частности, она должна включать достаточно полную и понятную пользователям документацию, возможно, специальную документацию для администраторов, а также набор документов для обучения работе с программой;

– программа должна обладать высокой производительностью, высокой реактивностью или удовлетворять другим требованиям, в противном случае ее использование по назначению (на реальных данных) может привести к значимым для пользователей потерям;

– программа должна обладать высокой надежностью. Неправильная работа программы может нанести ощутимый ущерб пользователям и другим организациям и лицам, даже если сбои происходят не слишком часто;

– для выполнения своих задач программа должна удовлетворять требованиям совместимости, переносимости и интеграции с другими программами и программно-аппаратными системами и обеспечивать работу на разных платформах;

– пользователи, работающие с программой, могут приобретать дополнительные выгоды от того, что программа развивается, в нее вносятся новые функции и устраняются ошибки. Поэтому необходимо наличие проектной документации, позволяющей развивать ее, возможно, вовсе не тем разработчикам, которые ее создавали, без больших затрат на обратную разработку (реинжиниринг);

– в разработку программы вовлечено значительное количество людей (десятки и сотни человек). Большую программу практически невозможно написать с первой попытки, с небольшими усилиями и в одиночку;

– большая программа имеет намного большее количество ее возможных пользователей по сравнению с небольшими программами и еще больше тех лиц, деятельность которых будет так или иначе затронута ее работой и результатами.

Более подробно теоретические сведения и методики изложены в [1].

Практическая работа № 2. Жизненный цикл программного обеспечения

Цель работы: изучить жизненный цикл программного обеспечения.

План практической работы

Вводная часть.

Задание на практическое занятие: жизненный цикл программного обеспечения.

Основная часть.

Список обсуждаемых вопросов:

– жизненный цикл программного обеспечения;

– распределение финансовых и временных затрат на реализацию каждого из этапов разработки программного обеспечения;

– обследование системы, общение с заказчиком, планирование разработки, составление технического задания;

– детальный анализ предметной области, принятие окончательного решения о необходимости создания информационной системы, проектирование общей архитектуры системы, выбор метода проектирования.

Заключительная часть.

Подведение общих итогов занятия.

Ответы на вопросы студентов.

Выдача рекомендаций.

Контрольные вопросы

1 Что такое жизненный цикл программного обеспечения?
2 Как распределяются затраты на разработку программного обеспечения?
3 Как проводится анализ предметной области?

Краткие теоретические сведения

Жизненный цикл программного обеспечения (Software Life Cycle Model) – это период времени, который начинается с момента принятия решения о создании программного продукта и заканчивается в момент его полного изъятия из эксплуатации. Этот цикл – процесс построения и развития ПО.

Модели жизненного цикла программного обеспечения.

Жизненный цикл можно представить в виде моделей. В настоящее время наиболее распространенными являются: каскадная, инкрементная (поэтапная модель с промежуточным контролем) и спиральная модели жизненного цикла.

Каскадная модель (англ. «waterfall model») – модель процесса разработки программного обеспечения, жизненный цикл которой выглядит как поток, последовательно проходящий фазы анализа требований, проектирования, реализации, тестирования, интеграции и поддержки.

Процесс разработки реализуется с помощью упорядоченной последовательности независимых шагов. Модель предусматривает, что каждый последующий шаг начинается после полного завершения выполнения предыдущего шага. На всех шагах модели выполняются вспомогательные и организационные процессы и работы, включающие управление проектом, оценку и управление качеством, верификацию и аттестацию, менеджмент конфигурации, разработку документации. В результате завершения шагов формируются промежуточные продукты, которые не могут изменяться на последующих шагах.

Жизненный цикл традиционно разделяют на следующие основные этапы:

– анализ требований;
– проектирование;
– кодирование (программирование);
– тестирование и отладка;
– эксплуатация и сопровождение.

Достоинства модели:

– стабильность требований в течение всего жизненного цикла разработки;
– на каждой стадии формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
– определенность и понятность шагов модели и простота её применения;
– выполняемые в логической последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие ресурсы (денежные, материальные и людские).

Каскадная модель хорошо зарекомендовала себя при построении относительно простых ПО, когда в самом начале разработки можно достаточно точно и полно сформулировать все требования к продукту.

Недостатки модели:

– сложность чёткого формулирования требований и невозможность их динамического изменения на протяжении полного жизненного цикла;
– низкая гибкость в управлении проектом;
– последовательность линейной структуры процесса разработки, в ре-

зультате возврат к предыдущим шагам для решения возникающих проблем приводит к увеличению затрат и нарушению графика работ;

– непригодность промежуточного продукта для использования;

– невозможность гибкого моделирования уникальных систем;

– позднее обнаружение проблем, связанных со сборкой, в связи с одновременной интеграцией всех результатов в конце разработки;

– недостаточное участие пользователя в создании системы – в самом начале (при разработке требований) и в конце (во время приёмочных испытаний);

– пользователи не могут убедиться в качестве разрабатываемого продукта до окончания всего процесса разработки. Они не имеют возможности оценить качество, т. к. нельзя увидеть готовый продукт разработки;

– у пользователя нет возможности постепенно привыкнуть к системе. Процесс обучения происходит в конце жизненного цикла, когда ПО уже запущено в эксплуатацию;

– каждая фаза является предпосылкой для выполнения последующих действий, что превращает такой метод в рискованный выбор для систем, не имеющих аналогов, т. к. он не поддается гибкому моделированию.

Реализовать каскадную модель жизненного цикла затруднительно ввиду сложности разработки ПС без возвратов к предыдущим шагам и изменения их результатов для устранения возникающих проблем.

Область применения каскадной модели.

Ограничение области применения каскадной модели определяется её недостатками. Её использование наиболее эффективно в следующих случаях:

– при разработке проектов с четкими, неизменяемыми в течение жизненного цикла требованиями, понятными реализацией и техническими методиками;

– при разработке проекта, ориентированного на построение системы или продукта такого же типа, как уже разрабатывались разработчиками ранее;

– при разработке проекта, связанного с созданием и выпуском новой версии уже существующего продукта или системы;

– при разработке проекта, связанного с переносом уже существующего продукта или системы на новую платформу;

– при выполнении больших проектов, в которых задействовано несколько больших команд разработчиков.

Инкрементная модель (англ. «increment» – увеличение, приращение) подразумевает разработку программного обеспечения с линейной последовательностью стадий, но в несколько инкрементов (версий), т. е. с запланированным улучшением продукта за все время, пока жизненный цикл разработки ПО не подойдет к окончанию.

Разработка программного обеспечения ведется итерациями с циклами обратной связи между этапами. Межэтапные корректировки позволяют учитывать реально существующее взаимовлияние результатов разработки на различных этапах, время жизни каждого из этапов растягивается на весь период разработки.

В начале работы над проектом определяются все основные требования к системе, подразделяются на более и менее важные. После чего выполняется

разработка системы по принципу приращений так, чтобы разработчик мог использовать данные, полученные в ходе разработки ПО. Каждый инкремент должен добавлять системе определенную функциональность. При этом выпуск начинают с компонентов с наивысшим приоритетом. Когда части системы определены, берут первую часть и начинают её детализировать, используя для этого наиболее подходящий процесс. В то же время можно уточнять требования и для других частей, которые в текущей совокупности требований данной работы были заморожены. Если есть необходимость, можно вернуться позже к этой части. Если часть готова, она поставляется клиенту, который может использовать её в работе. Это позволит клиенту уточнить требования для следующих компонентов. Затем занимаются разработкой следующей части системы. Ключевые этапы этого процесса – простая реализация подмножества требований к программе и совершенствование модели в серии последовательных релизов до тех пор, пока не будет реализовано ПО во всей полноте.

Жизненный цикл данной модели характерен при разработке сложных и комплексных систем, для которых имеется четкое видение (как со стороны заказчика, так и со стороны разработчика) того, что собой должен представлять конечный результат. Разработка версиями ведется в силу разного рода причин:

– отсутствия у заказчика возможности сразу профинансировать весь дорогостоящий проект;

– отсутствия у разработчика необходимых ресурсов для реализации сложного проекта в сжатые сроки;

– требований поэтапного внедрения и освоения продукта конечными пользователями. Внедрение всей системы сразу может вызвать у её пользователей неприятие и только «затормозить» процесс перехода на новые технологии. Образно говоря, они могут просто «не переварить большой кусок, поэтому его надо измельчить и давать по частям».

Достоинства и недостатки этой модели (стратегии) такие же, как и у каскадной (классической модели жизненного цикла). Но в отличие от классической стратегии заказчик может раньше увидеть результаты. Уже по результатам разработки и внедрения первой версии он может незначительно изменить требования к разработке, отказаться от нее или предложить разработку более совершенного продукта с заключением нового договора.

Достоинства модели:

– затраты, которые получаются в связи с изменением требований пользователей, уменьшаются, повторный анализ и совокупность документации значительно сокращаются по сравнению с каскадной моделью;

– легче получить отзывы от клиента о проделанной работе – клиенты могут озвучить свои комментарии в отношении готовых частей и могут видеть, что уже сделано, т. к. первые части системы являются прототипом системы в целом;

– у клиента есть возможность быстро получить и освоить программное обеспечение – клиенты могут получить реальные преимущества от системы раньше, чем это было бы возможно с каскадной моделью.

Недостатки модели:

– менеджеры должны постоянно измерять прогресс процесса, в случае

быстрой разработки не стоит создавать документы для каждого минимального изменения версии;

– структура системы имеет тенденцию к ухудшению при добавлении новых компонентов – постоянные изменения нарушают структуру системы. Чтобы избежать этого, требуется дополнительное время и деньги на рефакторинг. Плохая структура делает программное обеспечение сложным и дорогостоящим для последующих изменений. А прерванный жизненный цикл ПО приводит еще к большим потерям.

Схема не позволяет оперативно учитывать возникающие изменения и уточнения требований к ПО. Согласование результатов разработки с пользователями производится только в точках, планируемых после завершения каждого этапа работ, а общие требования к ПО зафиксированы в виде технического задания на всё время её создания. Таким образом, пользователи зачастую получают ПП, не удовлетворяющий их реальным потребностям.

Спиральная модель: жизненный цикл – на каждом витке спирали выполняется создание очередной версии продукта, уточняются требования проекта, определяется его качество и планируются работы следующего витка. Особое внимание уделяется начальным этапам разработки – анализу и проектированию, где реализуемость тех или иных технических решений проверяется и обосновывается посредством создания прототипов.

Спиральная модель жизненного цикла.

Данная модель представляет собой процесс разработки программного обеспечения, сочетающий в себе как проектирование, так и постадийное прототипирование с целью сочетания преимуществ восходящей и нисходящей концепции, делающей упор на начальные этапы жизненного цикла: анализ и проектирование. Отличительной особенностью этой модели является специальное внимание рискам, влияющим на организацию жизненного цикла.

На этапах анализа и проектирования реализуемость технических решений и степень удовлетворения потребностей заказчика проверяется путем создания прототипов. Каждый виток спирали соответствует созданию работоспособного фрагмента или версии системы. Это позволяет уточнить требования, цели и характеристики проекта, определить качество разработки, спланировать работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который удовлетворяет действительным требованиям заказчика и доводится до реализации.

Жизненный цикл на каждом витке спирали – могут применяться разные модели процесса разработки ПО. В конечном итоге на выходе получается готовый продукт. Модель сочетает в себе возможности модели прототипирования и водопадной модели. Разработка итерациями отражает объективно существующий спиральный цикл создания системы. Неполное завершение работ на каждом этапе позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем. Главная задача – как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Достоинства модели:

– позволяет быстрее показать пользователям системы работоспособный продукт, тем самым, активизируя процесс уточнения и дополнения требований;

– допускает изменение требований при разработке программного обеспечения, что характерно для большинства разработок, в том числе и типовых;

– в модели предусмотрена возможность гибкого проектирования, поскольку в ней воплощены преимущества каскадной модели, и в то же время разрешены итерации по всем фазам этой же модели;

– позволяет получить более надежную и устойчивую систему. По мере развития программного обеспечения ошибки и слабые места обнаруживаются и исправляются на каждой итерации;

– эта модель разрешает пользователям активно принимать участие при планировании, анализе рисков, разработке, а также при выполнении оценочных действий;

– уменьшаются риски заказчика. Заказчик может с минимальными для себя финансовыми потерями завершить развитие неперспективного проекта;

– обратная связь по направлению от пользователей к разработчикам выполняется с высокой частотой и на ранних этапах модели, что обеспечивает создание нужного продукта высокого качества.

Недостатки модели:

– если проект имеет низкую степень риска или небольшие размеры, модель может оказаться дорогостоящей. Оценка рисков после прохождения каждой спирали связана с большими затратами;

– жизненный цикл модели имеет усложненную структуру, поэтому может быть затруднено её применение разработчиками, менеджерами и заказчиками;

– спираль может продолжаться до бесконечности, поскольку каждая ответная реакция заказчика на созданную версию может порождать новый цикл, что отдаляет окончание работы над проектом;

– большое количество промежуточных циклов может привести к необходимости в обработке дополнительной документации;

– использование модели может оказаться дорогостоящим и даже недопустимым по средствам, т. к. время, затраченное на планирование, повторное определение целей, выполнение анализа рисков и прототипирование, может быть чрезмерным;

– могут возникнуть затруднения при определении целей и стадий, указывающих на готовность продолжать процесс разработки на следующей итерации.

Основная проблема спирального цикла – определение момента перехода на следующий этап. Для её решения вводятся временные ограничения на каждый из этапов жизненного цикла и переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. Планирование производится на основе статистических данных, полученных в предыдущих проектах и личного опыта разработчиков.

Область применения спиральной модели.

Применение спиральной модели целесообразно в следующих случаях:

– при разработке проектов, использующих новые технологии;
– при разработке новой серии продуктов или систем;
– при разработке проектов с ожидаемыми существенными изменениями или дополнениями требований;
– для выполнения долгосрочных проектов;
– при разработке проектов, требующих демонстрации качества и версий системы или продукта через короткий период времени;
– при разработке проектов, для которых необходим подсчет затрат, связанных с оценкой и разрешением рисков.

Практическая работа № 3. Обзор методологий проектирования программных продуктов

Цель работы: изучить методологии проектирования программных продуктов.

План практической работы

Вводная часть.
Задание на практическое занятие: обзор методологий проектирования программных продуктов.
Основная часть.
Список обсуждаемых вопросов:
– обзор методологий проектирования программных продуктов;
– каскадные и итеративные технологии. Критичность и маштабность программных проектов.
Заключительная часть.
Подведение общих итогов занятия.
Ответы на вопросы студентов.
Выдача рекомендаций.

Контрольные вопросы

1 Какие существуют методологии разработки программного обеспечения?
2 Что такое каскадные методологии разработки программного обеспечения?
3 Что такое итеративные методологии разработки программного обеспечения?

Краткие теоретические сведения

Методология – это система принципов, а также совокупность идей, понятий, методов, способов и средств, определяющих стиль разработки программного обеспечения.
Методология – это реализация стандарта. Сами стандарты лишь говорят о том, что должно быть, оставляя свободу выбора и адаптации.

Конкретные вещи реализуется через выбранную методологию. Именно она определяет, как будет выполняться разработка. Существует много успешных методологий создания программного обеспечения. Выбор конкретной методологии зависит от размера команды, от специфики и сложности проекта, от стабильности и зрелости процессов в компании и от личных качеств сотрудников.

Методологии представляют собой ядро теории управления разработкой программного обеспечения. К существующей классификации в зависимости от используемой в ней модели жизненного цикла (водопадные и итерационные методологии) добавилась более общая классификация на прогнозируемые и адаптивные методологии.

Прогнозируемые методологии фокусируются на детальном планировании будущего. Известны запланированные задачи и ресурсы на весь срок проекта. Команда с трудом реагирует на возможные изменения. План оптимизирован исходя из состава работ и существующих требований. Изменение требований может привести к существенному изменению плана, а также дизайна проекта. Часто создается специальный комитет по «управлению изменениями», чтобы в проекте учитывались только самые важные требования.

Адаптивные методологии нацелены на преодоление ожидаемой неполноты требований и их постоянного изменения. Когда меняются требования, команда разработчиков тоже меняется. Команда, участвующая в адаптивной разработке, с трудом может предсказать будущее проекта. Существует точный план лишь на ближайшее время. Более удаленные во времени планы существуют лишь как декларации о целях проекта, ожидаемых затратах и результатах.

SCRUM – методология, предназначенная для небольших команд (до 10 человек). Весь проект делится на итерации (спринты) продолжительностью 30 дней каждый. Выбирается список функций системы, которые планируется реализовать в течение следующего спринта. Самые важные условия – неизменность выбранных функций во время выполнения одной итерации и строгое соблюдение сроков выпуска очередного релиза, даже если к его выпуску не удастся реализовать весь запланированный функционал. Руководитель разработки проводит ежедневные 20-минутные совещания, которые так и называют – scrum, результатом которых являются определение функций системы, реализованных за предыдущий день, возникшие сложности и план на следующий день. Такие совещания позволяют постоянно отслеживать ход проекта, быстро выявлять возникшие проблемы и оперативно на них реагировать.

KANBAN – гибкая методология разработки программного обеспечения, ориентированная на задачи.

Основные правила:

– визуализация разработки;
– разделение работы на задачи;
– использование отметок о положении задачи в разработке;
– ограничение работ, выполняющихся одновременно, на каждом этапе разработки;
– измерение времени цикла (среднее время на выполнение одной задачи) и оптимизация процесса.

Преимущества KANBAN:

– уменьшение числа параллельно выполняемых задач значительно уменьшает время выполнения каждой отдельной задачи;

– быстрое выявление проблемных задач;

– вычисление времени на выполнение усредненной задачи.

Dynamic system development methOD появился в результате работы консорциум из 17 английских компаний. Целая организация занимается разработкой пособий по этой методологии, организацией учебных курсов, программ аккредитации и т. п. Кроме того, ценность DSDM имеет денежный эквивалент.

Все начинается с изучения осуществимости программы и области ее применения. В первом случае, Вы пытаетесь понять, подходит ли DSDM для данного проекта. Изучать область применения программы предполагается на короткой серии семинаров, где программисты узнают о той сфере бизнеса, для которой им предстоит работать. Здесь же обсуждаются основные положения, касающиеся архитектуры будущей системы, и план проекта.

Далее процесс делится на три взаимосвязанных цикла: цикл функциональной модели отвечает за создание аналитической документации и прототипов, цикл проектирования и конструирования – за приведение системы в рабочее состояние, и наконец, последний цикл – цикл реализации – обеспечивает развертывание программной системы.

Базовые принципы, на которых строится DSDM, это активное взаимодействие с пользователями, частые выпуски версий, самостоятельность разработчиков в принятии решений и тестирование в течение всего цикла работ. Как и большинство других гибких методологий, DSDM использует короткие итерации, продолжительностью от двух до шести недель каждая. Особый упор делается на высоком качестве работы и адаптируемости к изменениям в требованиях.

Microsoft solutions framework – методология разработки программного обеспечения, предложенная корпорацией Microsoft. MSF опирается на практический опыт Microsoft и описывает управление людьми и рабочими процессами в процессе разработки решения.

Базовые концепции и принципы модели процессов MSF:

– единое видение проекта – все заинтересованные лица и просто участники проекта должны чётко представлять конечный результат, всем должна быть понятна цель проекта;

– управление компромиссами – поиск компромиссов между ресурсами проекта, календарным графиком и реализуемыми возможностями;

– гибкость – готовность к изменяющимся проектным условиям;

– концентрация на бизнес-приоритетах – сосредоточенность на той отдаче и выгоде, которую ожидает получить потребитель решения;

– поощрение свободного общения внутри проекта;

– создание базовых версии – фиксация состояния любого проектного артефакта, в том числе программного кода, плана проекта, руководства пользователя, настройки серверов и последующее эффективное управление изменениями, аналитика проекта.

MSF предлагает проверенные методики для планирования, проектирования, разработки и внедрения успешных IT-решений. Благодаря своей гибкости, масштабируемости и отсутствию жестких инструкций MSF способен удовлетворить нужды организации или проектной группы любого размера. Методология MSF состоит из принципов, моделей и дисциплин по управлению персоналом, процессами, технологическими элементами и связанными со всеми этими факторами вопросами, характерными для большинства проектов.

Rational unified process – методология разработки программного обеспечения, созданная компанией Rational Software.

В основе методологии лежат шесть основных принципов:

1) компонентная архитектура, реализуемая и тестируемая на ранних стадиях проекта;

2) работа над проектом в сплочённой команде, ключевая роль в которой принадлежит архитекторам;

3) ранняя идентификация и непрерывное устранение возможных рисков;

4) концентрация на выполнении требований заказчиков к исполняемой программе;

5) ожидание изменений в требованиях, проектных решениях и реализации в процессе разработки;

6) постоянное обеспечение качества на всех этапах разработки проекта.

Использование методологии RUP направлено на итеративную модель разработки. Особенность методологии состоит в том, что степень формализации может меняться в зависимости от потребностей проекта. Можно по окончании каждого этапа и каждой итерации создавать все требуемые документы и достигнуть максимального уровня формализации, а можно создавать только необходимые для работы документы, вплоть до полного их отсутствия. За счет такого подхода к формализации процессов методология является достаточно гибкой и широко популярной. Данная методология применима как в небольших и быстрых проектах, где за счет отсутствия формализации требуется сократить время выполнения проекта и расходы, так и в больших и сложных проектах, где требуется высокий уровень формализма, например, с целью дальнейшей сертификации продукта. Это преимущество дает возможность использовать одну и ту же команду разработчиков для реализации, различных по объему и требованиям.

Более подробно теоретические сведения и методики изложены в [1].

Практическая работа № 4. Технологии быстрой разработки программного обеспечения

Цель работы: изучить технологии быстрой разработки программного обеспечения.

План практической работы

Вводная часть.

Задание на практическое занятие: обзор технологий быстрой разработки программного обеспечения.

Основная часть.

Список обсуждаемых вопросов:

– технология экстремального программирования;

– SCRUM-технология;

– преимущества и недостатки технологий быстрой разработки программного обеспечения;

– организация коллективной работы над проектом при использовании технологий быстрой разработки.

Заключительная часть.

Подведение общих итогов занятия.

Ответы на вопросы студентов.

Выдача рекомендаций.

Контрольные вопросы

1 Какие существуют технологии экстремальномого программирования?

2 Что такое SCRUM-технология?

3 Каковы преимущества и недостатки технологий быстрой разработки программного обеспечения?

Краткие теоретические сведения

Rapid Application Development (RAD) – это жизненный цикл процесса проектирования, созданный для достижения более высоких скорости разработки и качества ПО, чем это возможно при традиционном подходе к проектированию.

RAD предполагает, что разработка ПО осуществляется небольшой командой разработчиков за срок порядка трех-четырех месяцев путем использования инкрементного прототипирования с применением инструментальных средств визуального моделирования и разработки. Технология RAD предусматривает активное привлечение заказчика уже на ранних стадиях – обследование организации, выработка требований к системе. Причины популярности RAD вытекают из тех преимуществ, которые обеспечивает эта технология.

Наиболее существенными из них являются:

– высокая скорость разработки;
– низкая стоимость;
– высокое качество.

Последнее из указанных свойств подразумевает полное выполнение требований заказчика как функциональных, так и нефункциональных, с учетом их возможных изменений в период разработки системы, а также получение качественной документации, обеспечивающей удобство эксплуатации и сопровождения системы. Это означает, что дополнительные затраты на сопровождение сразу после поставки будут значительно меньше. Таким образом, полное время от начала разработки до получения приемлемого продукта при использовании этого метода значительно сокращается.

Применение технологии RAD целесообразно, когда:

– требуется выполнение проекта в сжатые сроки (90 дней). Быстрое выполнение проекта позволяет создать систему, отвечающую требованиям сегодняшнего дня. Если система проектируется долго, то весьма высока вероятность, что за это время существенно изменятся фундаментальные положения, регламентирующие деятельность организации, т. е. система морально устареет еще до завершения ее проектирования;

– нечетко определены требования к ПО. В большинстве случаев заказчик весьма приблизительно представляет себе работу будущего программного продукта и не может четко сформулировать все требования к ПО. Требования могут быть вообще не определены к началу проекта либо могут изменяться по ходу его выполнения;

– проект выполняется в условиях ограниченности бюджета. Разработка ведется небольшими RAD-группами в короткие сроки, что обеспечивает минимум трудозатрат и позволяет вписаться в бюджетные ограничения.

– интерфейс пользователя (GUI) есть главный фактор. Нет смысла заставлять пользователя рисовать картинки. RAD-технология дает возможность продемонстрировать интерфейс в прототипе, причем достаточно скоро после начала проекта;

– проект большой, но поддается разделению на более мелкие функциональные компоненты. Если предполагаемая система велика, необходимо, чтобы ее можно было разбить на мелкие части, каждая из которых обладает четкой функциональностью. Они могут выпускаться последовательно или параллельно (в последнем случае привлекается несколько RAD-групп);

– ПО не обладает большой вычислительной сложностью.

RAD-технология не является универсальной, т. е. ее применение целесообразно не всегда. Например, в проектах, где требования к программному продукту четко определены и не должны меняться, вовлечение заказчика в процесс разработки не требуется и более эффективной может быть иерархическая разработка (каскадный метод). То же касается проектов, ПО, сложность которых определяется необходимостью реализации сложных алгоритмов, а роль и объем пользовательского интерфейса невелик.

Принципы организации RAD.

Принципы RAD-технологии направлены на обеспечение трех основных ее преимуществ – высокой скорости разработки, низкой стоимости и высокого качества. Достигнуть высокого качества программного продукта весьма непросто и одна из главных причин возникающих трудностей заключается в том, что разработчик и заказчик видят предмет разработки ПО по-разному.

Главная идея RAD-технологии состоит в том, чтобы как можно быстрее донести до заказчика результаты разработки, пусть и не в полном виде. Например, реализация только пользовательского интерфейса и предъявление его заказчику позволяет уже на ранней стадии разработки получить замечания по экранным и отчетным формам и внести необходимые коррективы. В этом случае значительно возрастает вероятность успеха проекта, т. е. возникает уверенность в том, что конечный продукт будет делать именно то, что ожидает заказчик. Кроме того, не следует забывать и тот факт, что разница стоимости ошибки определения требований в начале проекта и в конце равна 1 : 200.

Основные принципы RAD можно сформулировать следующим образом:

– работа ведется группами. Типичный состав группы – руководитель, аналитик, два программиста, технический писатель. Если проект сложный, то для него может быть выделено несколько RAD-групп. Разработка проекта выполняется в условиях тесного взаимодействия между разработчиками и Заказчиком;

– разработка базируется на моделях. Моделирование позволяет оценить проект и выполнить его декомпозицию на составные части, каждая из которых может разрабатываться отдельной RAD-группой;

– итерационное прототипирование. Разработка системы и предъявление ее заказчику осуществляется в виде последовательности развиваемых прототипов. Любой из прототипов реализует определенную часть функциональности, требуемой от конечного продукта. При этом каждый последующий прототип включает всю функциональность, реализованную в предыдущем прототипе, с добавлением новой. Число прототипов определяется на основе учета разных параметров – размера проекта, анализа рисков, пожеланий заказчика и т. д. Традиционно для проектов ПО средней сложности разрабатываются три прототипа. Первый содержит весь пользовательский интерфейс с нулевой функциональностью. Он дает возможность собрать замечания заказчика и после их устранения утвердить у него экранные и отчетные формы. Второй прототип содержит реализованную на 70…80 % функциональность системы, третий – полностью реализованную функциональность. RAD-группа всегда работает только над одним прототипом. Это обеспечивает единство целей, лучшую наблюдаемость и управляемость процессом разработки, что в итоге повышает качество конечного продукта. Соответственно используемые инструментальные средства должны обеспечивать групповую разработку и конфигурационное управление проектом;

– если проект сложный, то для него может быть выделено несколько RAD-групп. Большие системы разбиваются на подсистемы. Каждая подсистема разрабатывается независимой группой. Ключ успеха – правильное разбиение системы на подсистемы. Команды должны использовать общие стандарты. Обязательно финальное тестирование полной системы;

– обязательное использование инструментальных средств, автоматизирующих процесс разработки, и методик их использования, следствием чего является сокращение сроков разработки и повышение качества конечного продукта.

Принципы RAD применяются не только при реализации, но и распространяются на все этапы жизненного цикла, в частности на этап обследования организации, построения требований, анализ и дизайн.

Технология RAD обеспечивает:

– быстроту продвижения программного продукта на рынок;
– интерфейс, устраивающий пользователя;
– легкую адаптируемость проекта к изменяющимся требованиям;
– простоту развития функциональности системы.

Более подробно теоретические сведения и методики изложены в [1].

5 Практическая работа № 5. Использование унифицированного языка моделирования при проектировании программных систем

Цель работы: изучить возможности использования унифицированного языка моделирования при проектировании программных систем.

План практической работы

Вводная часть.

Задание на практическое занятие: использование унифицированного языка моделирования при проектировании программных систем.

Основная часть.

Список обсуждаемых вопросов:

– введение в UML;
– основные диаграммы унифицированного языка моделирования;
– диаграмма вариантов использования.

Заключительная часть.

Подведение общих итогов занятия.

Ответы на вопросы студентов.

Выдача рекомендаций.

Контрольные вопросы

1 Какие основные диаграммы унифицированного языка моделирования существуют?

2 Что такое UML?

3 Как применяются диаграммы вариантов использования?

Краткие теоретические сведения

Использование языка UML основывается на следующих общих принципах моделирования:

– абстрагирование – в модель следует включать только те элементы проектируемой системы, которые имеют непосредственное отношение к выполнению ей своих функций или своего целевого предназначения. Другие элементы опускаются, чтобы не усложнять процесс анализа и исследования модели;

– многомодельность – никакая единственная модель не может с достаточной степенью точности описать различные аспекты системы. Допускается описывать систему некоторым числом взаимосвязанных представлений, каждое из которых отражает определенный аспект её поведения или структуры;

– иерархическое построение – при описании системы используются различные уровни абстрагирования и детализации в рамках фиксированных представлений. При этом первое представление системы описывает её в наиболее общих чертах и является представлением концептуального уровня, а последующие уровни раскрывают различные аспекты системы с возрастающей степенью детализации вплоть до физического уровня. Модель физического уровня в языке UML отражает компонентный состав проектируемой системы с точки зрения ее реализации на аппаратурной и программной платформах конкретных производителей.

Сущности в UML.

В UML определены четыре типа сущностей: структурные, поведенческие, группирующие и аннотационные. Сущности являются основными объектно-ориентированными элементами языка, с помощью которых создаются модели.

Структурные сущности – это имена существительные в моделях на языке UML. Как правило, они представляют статические части модели, соответствующие концептуальным или физическим элементам системы. Примерами структурных сущностей являются «класс», «интерфейс», «кооперация», «прецедент», «компонент», «узел», «актер».

Поведенческие сущности являются динамическими составляющими модели UML. Это глаголы, которые описывают поведение модели во времени и в пространстве. Существует два основных типа поведенческих сущностей:

– взаимодействие – это поведение, суть которого заключается в обмене сообщениями между объектами в рамках конкретного контекста для достижения определенной цели;

– автомат – алгоритм поведения, определяющий последовательность состояний, через которые объект или взаимодействие проходят в ответ на различные события.

Группирующие сущности являются организующими частями модели UML. Это блоки, на которые можно разложить модель. Такая первичная сущность имеется в единственном экземпляре – это пакет.

Пакеты представляют собой универсальный механизм организации элементов в группы. В пакет можно поместить структурные, поведенческие и другие группирующие сущности. В отличие от компонентов, которые реально су-

ществуют во время работы программы, пакеты носят чисто концептуальный характер, т. е. существуют только в процессе разработки.

Аннотационные сущности – это пояснительные части модели UML: комментарии для дополнительного описания, разъяснения или замечания к любому элементу модели. Имеется только один базовый тип аннотационных элементов – примечание. Примечание используют, чтобы снабдить диаграммы комментариями или ограничениями, выраженными в виде неформального или формального текста.

Отношения в UML.

В языке UML определены следующие типы отношений: зависимость, ассоциация, обобщение и реализация. Эти отношения являются основными связующими конструкциями UML и также как сущности применяются для построения моделей.

Зависимость (dependency) – это семантическое отношение между двумя сущностями, при котором изменение одной из них, независимой, может повлиять на семантику другой, зависимой.

Ассоциация (association) – структурное отношение, описывающее совокупность смысловых или логических связей между объектами.

Обобщение (generalization) – это отношение, при котором объект специализированного элемента (потомок) может быть подставлен вместо объекта обобщенного элемента (предка). При этом, в соответствии с принципами объектно-ориентированного программирования, потомок (child) наследует структуру и поведение своего предка (parent).

Реализация (realization) является семантическим отношением между классификаторами, при котором один классификатор определяет обязательство, а другой гарантирует его выполнение. Отношение реализации встречаются в двух случаях:

- между интерфейсами и реализующими их классами или компонентами;
- между прецедентами и реализующими их кооперациями.

Общие механизмы UML.

Для точного описания системы в UML используются так называемые общие механизмы:

- спецификации (specifications);
- дополнения (adornments);
- деления (common divisions);
- расширения (extensibility mechanisms).

UML является не только графическим языком. За каждым графическим элементом его нотации стоит спецификация, содержащая текстовое представление соответствующей конструкции языка. Например, пиктограмме класса соответствует спецификация, которая описывает его атрибуты, операции и поведение, хотя визуально, на диаграмме, пиктограмма часто отражает только малую часть этой информации. Более того, в модели может присутствовать другое представление этого класса, отражающее совершенно иные его аспекты, но, тем не менее, соответствующее спецификации. Таким образом, графическая нота-

ция UML используются для визуализации системы, а с помощью спецификаций описывают ее детали.

Практически каждый элемент UML имеет уникальное графическое изображение, которое дает визуальное представление самых важных его характеристик. Нотация сущности «класс» содержит его имя, атрибуты и операции. Спецификация класса может содержать и другие детали, например, видимость атрибутов и операций, комментарии или указание на то, что класс является абстрактным. Многие из этих деталей можно визуализировать в виде графических или текстовых дополнений к стандартному прямоугольнику, который изображает класс.

При моделировании объектно-ориентированных систем существует определенное деление представляемых сущностей.

Во-первых, существует деление на классы и объекты. Класс – это абстракция, а объект – конкретное воплощение этой абстракции. В связи с этим, почти все конструкции языка характеризуются двойственностью «класс/объект». Так, имеются прецеденты и экземпляры прецедентов, компоненты и экземпляры компонентов, узлы и экземпляры узлов. В графическом представлении для объекта принято использовать тот же символ, что и для класса, а название подчеркивать.

Во-вторых, существует деление на интерфейс и его реализацию. Интерфейс декларирует обязательства, а реализация представляет конкретное воплощение этих обязательств и обеспечивает точное следование объявленной семантике. В связи с этим, почти все конструкции UML характеризуются двойственностью «интерфейс/реализация». Например, прецеденты реализуются кооперациями, а операции – методами.

Виды диаграмм UML.

Графические изображения моделей системы в UML называются диаграммами. В терминах языка UML определены следующие их виды:

- диаграмма вариантов использования или прецедентов (use case diagram);
- диаграмма классов (class diagram);
- диаграммы поведения (behavior diagrams);
- диаграмма состояний (statechart diagram);
- диаграмма деятельности (activity diagram);
- диаграммы взаимодействия (interaction diagrams);
- диаграмма последовательности (sequence diagram);
- диаграмма кооперации (collaboration diagram);
- диаграммы реализации (implementation diagrams);
- диаграмма компонентов (component diagram);
- даграмма развертывания (deployment diagram).

Каждая из этих диаграмм конкретизирует различные представления о модели системы. При этом, диаграмма вариантов использования представляет концептуальную модель системы, которая является исходной для построения всех остальных диаграмм. Диаграмма классов является логической моделью, отражающей статические аспекты структурного построения системы, а диаграммы поведения, также являющиеся разновидностями логической модели, отражают динамические аспекты её функционирования. Диаграммы реализации служат для представления

компонентов системы и относятся к ее физической модели.

Из перечисленных выше диаграмм некоторые служат для обозначения двух и более подвидов. В качестве же самостоятельных представлений используются следующие диаграммы: вариантов использования, классов, состояний, деятельности, последовательности, кооперации, компонентов и развертывания.

Для диаграмм языка UML существуют три типа визуальных обозначений, которые важны с точки зрения заключенной в них информации:

1) связи, которые представляются различными линиями на плоскости;
2) текст, содержащийся внутри границ отдельных геометрических фигур;
3) графические символы, изображаемые вблизи визуальных элементов диаграмм.

При графическом изображении диаграмм рекомендуется придерживаться следующих правил:

– каждая диаграмма должна быть законченным представлением некоторого фрагмента моделируемой предметной области;
– представленные на диаграмме сущности модели должны быть одного концептуального уровня;
– вся информация о сущностях должна быть явно представлена на диаграмме;
– диаграммы не должны содержать противоречивой информации;
– диаграммы не следует перегружать текстовой информацией;
– каждая диаграмма должна быть самодостаточной для правильной интерпретации всех ее элементов;
– количество типов диаграмм, необходимых для описания конкретной системы, не является строго фиксированным и определяется разработчиком;
– модели системы должны содержать только те элементы, которые определены.

Более подробно теоретические сведения и методики изложены в [1].

Практическая работа № 6. Использование унифицированного языка моделирования при проектировании программных систем

Цель работы: изучить возможности использования унифицированного языка моделирования при проектировании программных систем.

План практической работы

Вводная часть.

Задание на практическое занятие: использование унифицированного языка моделирования при проектировании программных систем.

Основная часть.

Список обсуждаемых вопросов:

– кооперативные диаграммы;

– диаграмма классов;
– диаграмма компонентов.

Заключительная часть.
Подведение общих итогов занятия.
Ответы на вопросы студентов.
Выдача рекомендаций.

Контрольные вопросы

1 Что такое кооперативные диаграммы?
2 Что такое диаграмма классов?
3 Что такое диаграмма компонентов?

Краткие теоретические сведения

Кооперативные диаграммы. Понятие кооперации (collaboration) является одним из фундаментальных понятий в языке UML. Оно служит для обозначения множества взаимодействующих с определенной целью объектов в общем контексте моделируемой системы. Цель самой кооперации состоит в том, чтобы специфицировать особенности реализации отдельных наиболее значимых операций в системе. Кооперация определяет структуру поведения системы в терминах взаимодействия участников этой кооперации.

Кооперация может быть представлена на двух уровнях:

1) на уровне спецификации – показывает роли классификаторов и роли ассоциаций в рассматриваемом взаимодействии;

2) на уровне примеров – указывает экземпляры и связи, образующие отдельные роли в кооперации.

Диаграмма кооперации уровня спецификации показывает роли, которые играют участвующие во взаимодействии элементы. Элементами кооперации на этом уровне являются классы и ассоциации, которые обозначают отдельные роли классификаторов и ассоциации между участниками кооперации.

Диаграмма кооперации уровня примеров представляется совокупностью объектов (экземпляры классов) и связей (экземпляры ассоциаций). При этом связи дополняются стрелками сообщений. На данном уровне показываются только релевантные объекты, т. е. имеющие непосредственное отношение к реализации операции или классификатора.

В кооперации уровня примеров определяются свойства, которые должны иметь экземпляры для того, чтобы участвовать в кооперации. Кроме свойств объектов на диаграмме кооперации также указываются ассоциации, которые должны иметь место между объектами кооперации. При этом вовсе не обязательно изображать все свойства или все ассоциации, поскольку на диаграмме кооперации присутствуют только роли классификаторов, но не сами классификаторы. Таким образом, в то время как классификатор требует полного описания всех своих экземпляров, роль классификатора требует описания только тех свойств и ассоциаций, которые необходимы для участия в отдельной кооперации.

Отсюда вытекает важное следствие. Одна и та же совокупность объектов может участвовать в различных кооперациях. При этом, в зависимости от рассматриваемой кооперации, могут изменяться как свойства отдельных объектов, так и связи между ними. Именно это отличает диаграмму кооперации от диаграммы классов, на которой должны быть указаны все свойства и ассоциации между элементами диаграммы.

Диаграмма кооперации уровня спецификации. Кооперация на уровне спецификации изображается на диаграмме пунктирным эллипсом, внутри которого записывается имя этой кооперации. Такое представление кооперации относится к отдельному варианту использования и детализирует особенности его последующей реализации. Символ эллипса кооперации соединяется отрезками пунктирной линии с каждым из участников этой кооперации, в качестве которых могут выступать объекты или классы. Каждая из этих пунктирных линий помечается ролью (role) участника. Роли соответствуют именам элементов в контексте всей кооперации. Эти имена трактуются как параметры, которые ограничивают спецификацию элементов при любом их появлении в отдельных представлениях модели.

Диаграммы классов используются при моделировании ПС наиболее часто. Они являются одной из форм статического описания системы с точки зрения ее проектирования, показывая ее структуру. Диаграмма классов не отображает динамическое поведение объектов изображенных на ней классов. На диаграммах классов показываются классы, интерфейсы и отношения между ними.

Класс – это основной строительный блок ПС. Это понятие присутствует и в ОО языках программирования, т. е. между классами UML и программными классами есть соответствие, являющееся основой для автоматической генерации программных кодов или для выполнения реинжиниринга. Каждый класс имеет название, атрибуты и операции. Класс на диаграмме показывается в виде прямоугольника, разделенного на три области: в верхней содержится название класса; в средней – описание атрибутов (свойств); в нижней – названия операций – услуг, предоставляемых объектами этого класса.

Атрибуты класса определяют состав и структуру данных, хранимых в объектах этого класса. Каждый атрибут имеет имя и тип, определяющий, какие данные он представляет. При реализации объекта в программном коде для атрибутов будет выделена память, необходимая для хранения всех атрибутов, и каждый атрибут будет иметь конкретное значение в любой момент времени работы программы. Объектов одного класса в программе может быть много, все они имеют одинаковый набор атрибутов, описанный в классе, но значения атрибутов у каждого объекта свои и могут изменяться в ходе выполнения программы.

Для каждого атрибута класса можно задать видимость (visibility). Эта характеристика показывает, доступен ли атрибут для других классов. В UML определены следующие уровни видимости атрибутов:

– открытый (public) – атрибут виден для любого другого класса (объекта);
– защищенный (protected) – атрибут виден для потомков данного класса;
– закрытый (private) – атрибут не виден внешними классами (объектами) и может использоваться только объектом, его содержащим.

Последнее значение позволяет реализовать свойство инкапсуляции данных. Например, объявив все атрибуты класса закрытыми, можно полностью скрыть от внешнего мира его данные, гарантируя отсутствие несанкционированного доступа к ним. Это позволяет сократить число ошибок в программе. При этом любые изменения в составе атрибутов класса никак не скажутся на остальной части программной системы.

Класс содержит объявления операций, представляющих собой определения запросов, которые должны выполнять объекты данного класса. Каждая операция имеет сигнатуру, содержащую имя операции, тип возвращаемого значения и список параметров, который может быть пустым. Реализация операции в виде процедуры – это метод, принадлежащий классу. Для операций, как и для атрибутов класса, определено понятие «видимость». Закрытые операции являются внутренними для объектов класса и недоступны из других объектов. Остальные образуют интерфейсную часть класса и являются средством интеграции класса в программную систему.

На диаграммах классов обычно показываются ассоциации и обобщения.

Каждая ассоциация несет информацию о связях между объектами внутри программной системы. Наиболее часто используются бинарные ассоциации, связывающие два класса. Ассоциация может иметь название, которое должно выражать суть отображаемой связи. Помимо названия, ассоциация может иметь такую характеристику, как множественность. Она показывает, сколько объектов каждого класса может участвовать в ассоциации. Множественность указывается у каждого конца ассоциации (полюса) и задается конкретным числом или диапазоном чисел. Множественность, указанная в виде звездочки, предполагает любое количество (в том числе, и ноль).

Ассоциация «включает» показывает, что набор может включать несколько различных товаров. В данном случае направленная ассоциация позволяет найти все виды товаров, входящие в набор, но не дает ответа на вопрос, входит ли товар данного вида в какой-либо набор.

Ассоциация сама может обладать свойствами класса, т. е. иметь атрибуты и операции. В этом случае она называется класс-ассоциацией и может рассматриваться как класс, у которого помимо явно указанных атрибутов и операций есть ссылки на оба связываемых ею класса.

Обобщение показывает, что набор товаров – это тоже товар, который может быть предметом заказа, продажи, поставки и т. д. Набор включает опись, в которой указывается, какие товары входят в набор, а класс-ассоциация «включает» определяет количество каждого вида товаров в наборе.

При создании диаграмм классов часто пользуются понятием «стереотип». В дальнейшем речь пойдет о стереотипах классов. Стереотип класса – это элемент расширения словаря UML, который обозначает отличительные особенности в использовании класса. Стереотип имеет название, которое задается в виде текстовой строки. При изображении класса на диаграмме стереотип показывается в верхней части класса в двойных угловых скобках.

Практическая работа № 7. Оценка качества программного обеспечения

Цель работы: изучить оценку качества программного обеспечения.

План практической работы

Вводная часть.
Задание на практическое занятие: оценка качества программного обеспечения.
Основная часть.
Список обсуждаемых вопросов:
- оценка качества программного обеспечения;
- методики оценки качества ПО;
- процессный подход к оценке качества ПО.

Заключительная часть.
Подведение общих итогов занятия.
Ответы на вопросы студентов.
Выдача рекомендаций.

Контрольные вопросы

1 Что такое оценка качества программного обеспечения?
2 Как используются методики оценки качества ПО?
3 Как применяется процессный подход к оценке качества ПО?

Краткие теоретические сведения

На современных компьютерах установлено множество разнообразного программного обеспечения. И хочется, чтобы оно было качественное, работоспособное, работало без сбоев и т. д. Рассмотрим определение «качества ПО» (Software Quality) в контексте международных стандартов:

1) качество программного обеспечения – это степень, в которой программное обеспечение обладает требуемой комбинацией свойств. (1061–1998 IEEE Standard for Software Quality Metrics Methodology);

2) качество программного средства – совокупность свойств программного средства (ПС), которые обусловливают его пригодность удовлетворять заданные или подразумеваемые потребности в соответствии с его назначением (ГОСТ 28806–90 *Качество программных средств. Термины и определения*).

Стандарт ISO 9126. На данный момент наиболее распространена и используется многоуровневая модель качества программного обеспечения, представленная в наборе стандартов ISO 9126. Основой регламентирования показателей качества систем является международный стандарт ISO 9126 *Информационная технология. Оценка программного продукта. Характеристики качества и руководство по их применению*. В этом стандарте описано многоуровневое распределение характеристик ПО. На верхнем уровне выделено шесть основных

характеристик качества ПО, каждую из которых определяют набором атрибутов, имеющих соответствующие метрики для последующей оценки.

Согласно этой модели, функциональность программного средства (functionality) – совокупность свойств ПС, определяемая наличием и конкретными особенностями набора функций, способных удовлетворять заданные или подразумеваемые потребности качества наряду с ее надежностью как технической системы. Надежность (Reliability) – способность ПО выполнять требуемые задачи в обозначенных условиях на протяжении заданного промежутка времени или указанное количество операций. Удобство использования программного средства (usability) – совокупность свойств ПС, характеризующая усилия, необходимые для его использования, и оценку результатов его использования заданным кругом пользователей ПС. Эффективность (Efficiency) – способность ПО обеспечивать требуемый уровень производительности в соответствии с выделенными ресурсами, временем и другими обозначенными условиями. Удобство сопровождения (Maintainability) – легкость, с которой ПО может анализироваться, тестироваться, изменяться для исправления дефектов, для реализации новых требований, для облегчения дальнейшего обслуживания и адаптироваться к именующемуся окружению. Портативность (Portability) – совокупность свойств ПС, характеризующая приспособленность для переноса из одной среды функционирования в другие.

Практическая работа № 8. Внедрение и сопровождение программных продуктов

Цель работы: изучение внедрения и сопровождения программных продуктов.

План практической работы

Вводная часть.
Задание: внедрение и сопровождение программных продуктов.
Основная часть.
Список обсуждаемых вопросов:

- планирование процесса внедрения программного продукта;
- основные задачи, решаемые на этапе внедрения;
- процесс устранения ошибок на этапе внедрения;
- документирование программного обеспечения;
- техническая поддержка пользователей на этапе сопровождения.

Заключительная часть.
Подведение общих итогов занятия.
Ответы на вопросы студентов.
Выдача рекомендаций.

Контрольные вопросы

1 Какие основные задачи решаются на этапе внедрения?
2 Что такое ошибка?
3 Что представляет из себя процесс сопровождения?
4 Как осуществляется техническая поддержка пользователей?

Краткие теоретические сведения

Под сопровождением программного обеспечения понимают процесс улучшения, оптимизации и устранения дефектов программного обеспечения после передачи в эксплуатацию.

Основные стандарты:

- ISO/IEC 14764 (2006 г., русский перевод стандарта 1999–2002);
- ISO/IEC 12207 (2008 г., русский перевод стандарта 2010 г.);
- ISO 20000;
- SWEBOK (2004 г.);
- ITIL v3 (2007 г., обновление – 2011 г.);
- COBIT v5 (2012 г.).

Процесс сопровождения является одной из фаз жизненного цикла программного обеспечения, следующей за передачей ПО в эксплуатацию, и завершается выводом его из эксплуатации. В ходе сопровождения в программу вносятся изменения, с тем, чтобы исправить обнаруженные в процессе использования дефекты и недоработки, для добавления новой функциональности, повышения удобства использования (юзабилити) и роста уровня использования ПО. По стандарту ISO/IEC 12207, этот процесс входит в пять основных процессов жизненного цикла (ЖЦ) ПО: приобретение, поставка, разработка, эксплуатация, сопровождение.

В общем случае процесс сопровождения состоит из следующих задач:

- устранение сбоев;
- улучшение дизайна;
- расширение функциональных возможностей;
- создание интерфейсов взаимодействия с другими (внешними) системами;
- адаптация (например, портирование) для возможности работы на другой (или обновленной) аппаратной платформе, применение новых системных возможностей, функционирование в среде обновленной телекоммуникационной инфраструктуры и т. п.;
- миграция унаследованного (legacy) программного обеспечения; вывод программного обеспечения из эксплуатации.

Список литературы

1 **Орлов, С. А.** Программная инженерия / С. А. Орлов. – Санкт-Петербург : Питер, 2016. – 640 с.

2 **Лионг, Б.** Практическая программная инженерия на основе учебного примера / Б. Лионг, Л. Мацяшек. – Москва : Бином, 2013. – 960 с.

3 **Гласс, Р.** Программирование и конфликты 2.0. Теория и практика программной инженерии / Р. Гласс. – Санкт-Петербург : Символ, 2009. – 240 с.

4 **Батоврин, В.** Толковый словарь по системной и программной инженерии / В. Батоврин. – Москва : ДМК Пресс, 2012. – 280 с.

5 **Орлов, С.** Технологии разработки программного обеспечения. Стандарт третьего поколения / С. Орлов, Б. Цилькер. – Санкт-Петербург : Питер, 2012. – 608 с.